Function Applicaitons in Panda

Chapter – 4

4.1 Introduction

In last chapter, we implemented different functions using Python pandas DataFrame. These are mostly Python is a great language for doing data analysis. Python panda supports number of data aggregation/ descriptive functions to analyze data. In this chapter we will learn Python pandas function applications.

4.2 .pipe() Function

Pipes take the output from one function and feed it to the first argument of the next function. **Pipe()** function performs the custom operation for the entire dataframe. Pipe can be thought of as a function chaining. The syntax is:

DataFrame.pipe(func, *args, **kwargs)

To apply **pipe()**, the first argument of the function must be the data set.
Here,

- func : A function which will be passed as first argument.
- args : positional arguments passed into func.
- kwargs : a dictionary of keyword arguments passed into func.

Let us create a DataFrame with following data:

```
>>> import pandas as pd
>>> Data = {'Name': ['Aashna', 'Simran', 'Jack', 'Raghu', 'Somya', 'Ronald'],
            'English': [87, 64, 58, 74, 87, 78],
            'Accounts': [76, 76, 68, 72, 82, 68],
            'Economics': [82, 69, 78, 67, 78, 68],
            'Bst': [72, 56, 63, 64, 66, 71],
            'IP': [78, 75, 82, 86, 67, 71]}
>>> df = pd.DataFrame(Data, columns=['Name', 'English', 'Accounts', 'Economics', 'Bst', 'IP'])
Or
For data: df = pd.read_csv('E:/IPSource_XII/IPXIIChap04/MResult.csv')
```

```
>>> df
```

	Name	English	Accounts	Economics	Bst	IP
0	Aashna	87	76	82	72	78
1	Simran	64	76	69	56	75
2	Jack	58	68	78	63	82
3	Raghu	74	72	67	64	86
4	Somya	87	82	78	66	67
5	Ronald	78	68	68	71	71

```
>>> df = df.set_index(['Name'])          # DataFrame indexed permanently
>>> df
```

Name	English	Accounts	Economics	Bst	IP
Aashna	87	76	82	72	78
Simran	64	76	69	56	75
Jack	58	68	78	63	82
Raghu	74	72	67	64	86
Somya	87	82	78	66	67
Ronald	78	68	68	71	71

Now using the above DataFrame **df**, apply the function **.pipe()** to add 2 marks with every numeric column:

```
# Create a user-define function to add two numbers
>>> def Add_Two(Data, aValue):
        return Data + aValue
>>> df.pipe(Add_Two, 2)
```

Name	English	Accounts	Economics	Bst	IP
Aashna	89	78	84	74	80
Simran	66	78	71	58	77
Jack	60	70	80	65	84
Raghu	76	74	69	66	88
Somya	89	84	80	68	69
Ronald	80	70	70	73	73

From the above command,

```
df.pipe(Add_Two, 2)
```

Notice that the first argument **Add_Two** is of the **.pipe()** is the data set. For example, **Add_Two** accepts two arguments **Add_Two(Data, aValue)**. As **Data** is the first parameter that takes in the data set, we can directly use **pipe()**. We only need to specify to pipe what's the name of the argument in the function that refers to the data set.

4.3 .apply() Function

The **.apply()** function applies a function to each element in the Series or an axis of the DataFrame, i.e., either row wise or column wise. We can use **.apply()** to send a single column or a single row to a function. Basically, we can use custom functions when applying on Series and also when operating on chunks of data frames in groupbys. This is useful when cleaning up data - converting formats, altering values, etc. This method takes as argument the following:

- a general or user defined function
- any other parameters that the function would take

The syntax is:

DataFrame.apply(func, axis=0, broadcast=None, raw=False, reduce=None, result_type=None, args=(), **kwds)

Here,

- **func:** is the name operations which will be apply to each column or row.
- **axis:** axis along which the function is applied:
  - 0 or 'index': apply function to each column.
  - 1 or 'columns': apply function to each row.

Let us see the following data set with two columns containing numeric data:

A	B
1	6
2	7
3	8
4	9
5	10
6	11

When we use **apply()** function, by default it will apply on each column. For example, let us create the dataframe and find the sum of each column using the aggregate function **sum** with **apply()**.

```
>>> Test = {'A': [1, 2, 3, 4, 5, 6],
            'B': [6, 7, 8, 9, 10, 11]}
>>> tdf = pd.DataFrame(Test, columns=['A', 'B'])
>>> tdf
   A   B
0  1   6
1  2   7
2  3   8
3  4   9
4  5  10
5  6  11
```

As the name suggests, the .apply() function applies a function along any axis of the DataFrame. For example, to find the sum of all values of each column:

```
>>> tdf[['A', 'B']].apply(sum)
```

The above command will returns the sum of all the values of column A and column B.

```
A   21
B   51
dtype: int64
```

Use .apply with axis=1 to send every single row to a function

Similarly, to find the sum of all values of each row, the command is:

```
>>> tdf[['A', 'B']].apply(sum, axis=1)     # axis = 1 applies to each row
0    7
1    9
2   11
3   13
4   15
5   17
dtype: int64
```

Let take another dataframe example with following data set for applying the **apply()** function.

```
>>> import pandas as pd
>>> dfA = pd.read_csv('E:/IPSource_XII/IPXIIChap04/Pizza.csv')
>>> dfA
```

	OrdNum	Size	Topping	Price
0	PZ001	Small	Margherita	356.65
1	PZ002	Large	Peppy Paneer	545.70
2	PZ003	Extra Large	Bell Pepper	756.90
3	PZ004	Extra Large	Mexican Green Wave	654.00
4	PZ005	Extra Large	Pepperoni chicken	632.00
5	PZ006	Large	Chicken Sausage	480.60
6	PZ007	Small	Peri-Peri chicken	375.00

Use .apply() to send a column of every row to a function

Let us apply the **.apply()** function to send a column of every row to a function to calculate 4% tax as a new column called **Taxes** with following expression:

Taxes = Price * 0.04

So, to implement the expression as a function for each row with previous DataFrame (dfA), create a function called **Tax_calc()** with following:

```
>>> # we create a function to retrieve
>>> def Tax_calc(Price):
        Taxes = Price * 0.04          # 4% tax
        return Taxes
>>> dfA['Taxes'] = dfA.Price.apply(Tax_calc)
```

Here, the function **.apply()** retrives the **Tax_calc()** function to calculate the taxes for **Price** column. The **Tacx_calc()** function accepts a argument **Tax_calc(Price)**. The resulted data is stored as a new column called **Taxes** with the existing DataFrame **dfA**. The result of new DataFrame is:

```
>>> print (dfA)
```

	OrdNum	Size	Topping	Price	Taxes
0	PZ001	Small	Margherita	356.65	14.266
1	PZ002	Large	Peppy Paneer	545.70	21.828
2	PZ003	Extra Large	Bell Pepper	756.90	30.276
3	PZ004	Extra Large	M Green Wave	654.00	26.160
4	PZ005	Extra Large	Pepperoni chicken	632.00	25.280
5	PZ006	Large	Chicken Sausage	480.60	19.224
6	PZ007	Small	Peri-Peri chicken	375.00	15.000

Let us apply the **.apply()** function to send every single row to a function to calculate a new column **Taxes**:

```
>>> # we create a function to retrieve
>>> def Tax_calc(row):                # A row is an user-define name contain all the column values
        return row['Price'] * 0.04    # The price value is used from the row
>>> dfA.apply(Tax_calc, axis=1)       # a series is displayed with taxes
0    14.266
1    21.828
2    30.276
3    26.160
4    25.280
5    19.224
6    15.000
dtype: float64
```

Or

If you want add it as new columns using **.apply()** function, then wirte the following:

```
>>> df['Taxes'] = df.apply(Tax_calc, axis=1)        # to add a new column
```

The above command will again create a DataFrame with a new **Taxes** column.

Using Arbitrary function with apply()

The axis parameter of **apply()** functions is useful to travel through columns and rows. When any arbitrary function like max, min, etc. are applied with **apply()** function, it travels the entire columns and rows. If the axis = 0, it will travel downwards of each column. Similarly, when axis = 1, it will travel row wise right.

In previous dataframe **dfA**, we have 5 columns and 7 rows. Let us find the maximum value of '**Price**' and '**Taxes**' column using **apply()** function.

```
>>> # apply() method travel axis=0
>>> dfA.loc[:, 'Price':'Taxes'].apply(max, axis=0)
```

which will display the maximum value of Price and Taxes column as given below:

```
Price     756.900
Taxes      30.276
dtype: float64
```

Similarly, to find row wise maximum values of '**Price**' and '**Taxes**' columns:

```
>>> dfA.loc[:, 'Price':'Taxes'].apply(max, axis=1)
0   356.65
1   545.70
2   756.90
3   654.00
4   632.00
5   480.60
6   375.00
dtype: float64
```

4..3.1 Using Lambda Function

Keyword **lambda** in python is used to create anonymous functions. Anonymous functions are those functions who are unnamed. That means you are defining a function without any name of the function. A **lambda** function is a shorthand way to define a quick function that you need once.

The basic syntax to create a **lambda** function is:

lambda arguments : expression

Lambda functions can have any number of **arguments** but only one **expression**. The expression is evaluated and returned. Lambda functions cannot contain any statements and it returns a function object which can be assigned to any variable.

For example:

```
>>> fun = lambda x: x*x
```

Here, in **lambda x: x*x**; **x** is an argument to the function and **x*x** is the expression which gets executed and its values is returned as output. When we call the function fun with an argument, it will pass it as x and perform the expression **x*x**.

For example,

```
>>> fun = lambda x: x*x
>>> sqr = fun(5)        # Output: 25
```

To check the type of the lambda function, type the following:

```
>>> type(fun)
<class 'function'>
```

Similarly, let us see another example to add two numbers using lambda function:

```
>>> SumTwo = lambda x, y : x + y
>>> print (SumTwo(10, 20))    # Output: 30
```

Here, in **lambda x, y: x + y**; **x** and **y** are arguments to the function and **x + y** is the expression which gets executed and its values is returned as output.

Also, the **lambda x, y: x + y** returns a function object which can be assigned to any variable, in this case function object is assigned to the **SumTwo** variable.

Using Lambda Function with .apply()

In previous section we use the **.apply()** function to find the tax for each row in DataFrame **dfA**. Instead of using the above function (**Tax_calc**), we can use a **lambda** function. Let us do the same with a **lambda** function:

```
>>> dfA.apply(lambda row: row[3] * 0.04, axis=1)
0    14.266
1    21.828
2    30.276
3    26.160
4    25.280
5    19.224
6    15.000
dtype: float64
```

Here, the **row** parameter takes enter **row** of dataframe **dfA** and **row[3]** is the **Price** column.

Let us see the difference between a normal **def** defined function and **lambda** function. This is a program that returns the **Taxes** of every single row to a function.

```
>>> def Tax_calc(row):                  # A row is an user-define name contain all the column values
        return row['Price'] * 0.04  # The price value is used from the row
```

Here, while using **def**, we needed to define a function with a name **Tax_calc** and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the return keyword.

On the other hand, in the **lambda** function, it does not include a "**return**" statement; it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all.

Example **Using previous dataframe dfA, write the commands for the following:**

(*a*) Write a function to display Topping column into capitalization form.

(*b*) Apply a toppingcapital function over the column 'Topping'.

Solution The commands are:

(*a*) Command to create a lambda function toppingcapital:
toppingcapital = lambda x: x.upper()

(*b*) Apply the toppingcapital function over the column 'Topping'
dfA['Topping'].apply(toppingcapital)

which will display the following:

```
0              MARGHERITA
1            PEPPY PANEER
2             BELL PEPPER
3     MEXICAN GREEN WAVE
4       PEPPERONI CHICKEN
5         CHICKEN SAUSAGE
6        PERI-PERI CHICKEN
Name: Topping, dtype: object
```

4.4 Aggregation (groupby)

Pandas DataFrames have a **.groupby()** method that works in the same way as the SQL **Group By**. The main objective of this function is to split the data into sets and then apply some functionality on each subset. The most important operations made available by a GroupBy are aggregate, filter, transform, and apply.

- **Splitting** the data into groups based on some criteria with the levels of a categorical variable. This is generally the simplest step. For examples, a DataFrame can be split up by rows(axis=0) or columns(axis=1) into groups.
- **Applying** a function to each group individually. A function is applied to each group using .agg() or .apply(). There are 3 classes of functions we might consider:
  - Aggregate – estimate/compute summary statistics (like counts, means) for each group. This will reduce the size of the data. Some examples:
    - Compute group sum(), max(), min(), mean(), etc.
    - Compute group sizes / count().
  - Transform – within group standardization, imputation using group values. The size of the data will not change.
    Some examples:
    - Standardize data (zscore) within a group.
    - Filling NAs within groups with a value derived from each group.
  - Filter – ignore rows that belong to a particular group. This discards some groups, according to a group-wise computation that evaluates True or False. Some examples:
    - Discard data that belongs to groups with only a few members.
    - Filter out data based on the group sum or mean.
  - A combination of these 3
- **Combining** the results into a data structure like series or DataFrame.

Let us see the following data set with two columns that how they are grouped.

Data

Product	Sales
Black	120
Red	130
Black	120
Green	110
Red	125
Green	132
Red	115
Black	136
Green	144
Red	165

Data with groupby Product

Product	Sales
Black	120
Black	120
Black	136
Green	110
Green	132
Green	144
Red	130
Red	125
Red	115
Red	165

Group result

Group function results				
sum	max	min	mean	count
376	**136**	**120**	**125.33**	**3**
386	**144**	**110**	**128.67**	**3**
535	**165**	**115**	**133.75**	**4**

When we apply the **.groupby()** method to a Dataframe object, it returns a **GroupBy** object, which is then assigned to the grouped single variable or **groupby** variable. An important thing to note about a pandas **GroupBy** object is that no splitting of the Dataframe has taken place at the point of creating the object. The **GroupBy** object simply has all of the information it needs about the nature of the grouping. No aggregation will take place until we explicitly call an aggregation function on the **GroupBy** object.

The syntax is:

DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False, **kwargs)

Here,

- by : Used to determine the groups for the groupby. If by is a function, it's called on each value of the object's index.
- axis: axis along which the function is applied:
  - 0 or 'index': apply function to each column.
  - 1 or 'columns': apply function to each row.
- level : If the axis is a MultiIndex (hierarchical), group by a particular level or levels.
- as_index : For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output. If you mention as_Index=False, then it does not show the GroupBy index column(s). A new numeric index will display. Otherwise all the as index columns will be displayed in as_Index=True.
- sort : Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.
- group_keys : When calling apply, add group keys to index to identify pieces.
- squeeze : Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

Before using the groupy function, let us create a dataframe with following data:

```
>>> import pandas as pd
>>> df = pd.read_csv('E:/IPSource_XII/IPXIIChap04/Stock.csv')
>>> df
```

	Category_Name	Item_Num	Unit_Price	Sales_Quantity
0	Television	T001_Panasonic	27800	8
1	Washing Machine	W003_Samsung	9699	4
2	Refrigerator	R001_LG	43800	6
3	Microwave	M001_LG	13600	8
4	Television	T002_Sony	42200	4
5	Air Conditioner	A001_LG	23500	11
6	Microwave	M002_Samsung	18750	4
7	Washing Machine	W001_IFB	32600	12
8	Television	T003_LG	32500	4
9	Refrigerator	R002_Samsung	23300	4
10	Air Conditioner	A002_Carrier	43700	6
11	Microwave	M003_LG	28750	5
12	Television	T004_Sony	65800	5
13	Washing Machine	W002_LG	24200	10
14	Air Conditioner	A003_Samsung	23500	11
15	Refrigerator	R003_Onida	23300	4

Using the above DataFrame (**df**) we create a grouping of categories and apply a function to the categories. For example, let us apply the **groupby()** function to group the data on **Category_Name** column.

```
# Create a groupBy object
>>> dfC = df.groupby('Category_Name')
```

Here, the **groupby()** function creates a groupby object called **dfC**. When we print the object, it will display the following:

```
>>> print (dfC)
<pandas.core.groupby.DataFrameGroupBy object at 0x07370430>
```

This grouped variable (**dfC**) is now a **GroupBy** object. It has not actually computed anything yet except for some intermediate data about the group key **df['key1']**. The idea is that this object has all of the information needed to then apply some operation to each of the groups. We can print information through iterate only.

View Groups

To view the groupby object result:

```
>>> print (df.groupby('Category_Name').groups)
{'Air Conditioner': Int64Index([5, 10, 14], dtype='int64'),
'Microwave': Int64Index([3, 6, 11], dtype='int64'),
```

```
'Refrigerator': Int64Index([2, 9, 15], dtype='int64'),
'Television': Int64Index([0, 4, 8, 12], dtype='int64'),
'Washing Machine': Int64Index([1, 7, 13], dtype='int64')}
```

From the above output, it displays the **Cagegory_Name** wise number of indexes from DataFrame **df**.

Or

We can use **list()** method to see the details of the GroupBy object values:

```
>>> list(df['Item_Num'].groupby(df['Category_Name']))
[('Air Conditioner', 5          A001_LG
10    A002_Carrier
14    A003_Samsung
Name: Item Num, dtype: object), ('Microwave', 3          M001_LG
6     M002_Samsung
11        M003_LG
Name: Item Num, dtype: object), ('Refrigerator', 2          R001_LG
9     R002_Samsung
15     R003_Onida
Name: Item Num, dtype: object), ('Television', 0     T001_Panasonic
4         T002_Sony
8          T003_LG
12        T004_Sony
Name: Item Num, dtype: object), ('Washing Machine', 1     W003_Samsung
7        W001_IFB
13        W002_LG
Name: Item Num, dtype: object)]
```

Printing First row of each Group

Let's print the first row each of each group of pandas dataframe using **dataframe.first()** method.

```
>>> dfC.first()
```

	Item_Num	Unit_Price	Sales_Quantity
Category_Name			
Air Conditioner	A001_LG	23500	11
Microwave	M001_LG	13600	8
Refrigerator	R001_LG	43800	6
Television	T001_Panasonic	27800	8
Washing Machine	W003_Samsung	9699	4

By default, the groupby object has the same label name as the group name.

Iterate over dataframe groups

Object returned by the call to groupby() function can be used as an iterator. In previous example, we created a GroupBy object called **dfC**. Let us use the **for** loop to iterate the object **dfC**:

```
>>> for key, group_df in dfC:
        print("The group for Cagegory Name '{}' has {} rows".format(key,len(group_df)))
```

which prints the following:

```
The group for Cagegory Name 'Air Conditioner' has 3 rows
The group for Cagegory Name 'Microwave' has 3 rows
The group for Cagegory Name 'Refrigerator' has 3 rows
The group for Cagegory Name 'Television' has 4 rows
The group for Cagegory Name 'Washing Machine' has 3 rows
```

From the above command:

- **key** contains the name of the grouped element i.e. 'Air Conditioner', 'Microwave', 'Refrigerator', 'Television', 'Washing Machine'
- **group_df** is a normal dataframe containing only the data referring to the key.

Select a Group

Using the **get_group()** method, we can select a single group. For example, to select a particular group called 'Television' from the GroupBy object **dfC**:

```
>>> print (dfC.get_group('Television'))
```

	Item_Num	Unit_Price	Sales_Quantity
0	T001_Panasonic	27800	8
4	T002_Sony	42200	4
8	T003_LG	32500	4
12	T004_Sony	65800	5

Grouping by One key

To produce a result, we can apply an aggregate to this **DataFrameGroupBy** object, which will perform the appropriate apply/combine steps to produce the desired result. Let us simple use the aggregate function **sum()** with the groupby key.

```
# Finding the values contained in the "Category_Name" group
>>> df.groupby('Category_Name').sum()
```

Category_Name	Unit_Price	Sales_Quantity
Air Conditioner	90700	28
Microwave	61100	17

Refrigerator	90400	14
Television	168300	21
Washing Machine	66499	26

Here, the aggregate function **sum()** calculates the **Category_Name** wise total of **Unit_Price** and **Sales_Quantity**. The **sum()** method is just one possibility here; you can apply any common Pandas or NumPy aggregation function, as well as any valid DataFrame operation.

Resetting GroupBy rows index

We can reset the grouped row index in pandas with **reset_index()** function to make the index start from 0. For example,

```
>>> df.groupby('Category_Name').sum().reset_index()
```

	Category_Name	Unit_Price	Sales_Quantity
0	Air Conditioner	90700	28
1	Microwave	61100	17
2	Refrigerator	90400	14
3	Television	168300	21
4	Washing Machine	66499	26

Ordering groupby results

We can sort the GroupBy object result using **sort_values()** method. For example, let find the total **Unit_Price** for each **Category_Name** into a new DataFrame **df1**:

```
>>> df1 = df.groupby('Category_Name')['Unit_Price'].sum().reset_index().sort_values(by='Unit_Price')
>>> df1
```

	Category_Name	Unit_Price
1	Microwave	61100
4	Washing Machine	66499
2	Refrigerator	90400
0	Air Conditioner	90700
3	Television	168300

Displaying SQL-style grouped output

The groupby method uses an option called as_index to display SQL_style grouped output. For example, to display SQL_style category_Name wise aggretage sum of data:

```
>>> df.groupby('Category_Name', as_index=False).sum()
```

	Category_Name	Unit_Price	Sales_Quantity
0	Air Conditioner	90700	28
1	Microwave	61100	17
2	Refrigerator	90400	14
3	Television	168300	21
4	Washing Machine	66499	26

Example: **Write the command to find the maximum Unit_Price of each Category_Name of DataFrame df.**

```
# Find the maximum Unti_Price of each Category_Name
>>> df.groupby('Category_Name').Unit_Price.max()
```

Category_Name	
Air Conditioner	43700
Microwave	28750
Refrigerator	43800
Television	65800
Washing Machine	32600

Name: Unit_Price, dtype: int64

Grouping by TWO key

To use two key in **groupby()** function, let us find the first grouping based on "**Category_Name**" within each item number we are grouping based on "**Item_Num**":

```
# Finding the values contained in the "Category_Name" group
>>> dfD = df.groupby(['Category_Name', 'Item_Num'])
Or
>>> dfD = df.groupby([df['Category_Name'], df['Item_Num']])
>>> dfD.first()
```

		Unit_Price	Sales_Quantity
Category_Name	**Item_Num**		
Air Conditioner	A001_LG	23500	11
	A002_Carrier	43700	6
	A003_Samsung	23500	11
Microwave	M001_LG	13600	8
	M002_Samsung	18750	4
	M003_LG	28750	5
Refrigerator	R001_LG	43800	6
	R002_Samsung	23300	4
	R003_Onida	23300	4

Television	T001_Panasonic	27800	8
	T002_Sony	42200	4
	T003_LG	32500	4
	T004_Sony	65800	5
Washing Machine	W001_IFB	32600	12
	W002_LG	24200	10
	W003_Samsung	9699	4

Column-wise aggregations – optimized statistical methods

For simple statistical aggregations (of numeric columns of a DataFrame) we can call methods like sum(), max(), min(), mean(), etc. Before applying the functions, let us create a new DataFrame called **dfN** by adding the new column '**Total_Amount**' with previous DataFrame **df**:

```
# Find the values contained in the "Category Name" group
>>> Amount = []  # a blank list
>>> for index, row in df.iterrows():
        Amt = row['Unit_Price'] * row['Sales_Quantity']        # Calculating a row value
        Amount.append(Amt)                  # append current amount to a list.
>>> dfN = df.assign(Total_Amount = Amount)   # A new column 'Total_Amount' created
>>> print (dfN)
```

	Category_Name	Item_Num	Unit_Price	Sales_Quantity	Total_Amount
0	Television	T001_Panasonic	27800	8	222400
1	Washing Machine	W003_Samsung	9699	4	38796
2	Refrigerator	R001_LG	43800	6	262800
3	Microwave	M001_LG	13600	8	108800
4	Television	T002_Sony	42200	4	168800
5	Air Conditioner	A001_LG	23500	11	258500
6	Microwave	M002_Samsung	18750	4	75000
7	Washing Machine	W001_IFB	32600	12	391200
8	Television	T003_LG	32500	4	130000
9	Refrigerator	R002_Samsung	23300	4	93200
10	Air Conditioner	A002_Carrier	43700	6	262200
11	Microwave	M003_LG	28750	5	143750
12	Television	T004_Sony	65800	5	329000
13	Washing Machine	W002_LG	24200	10	242000
14	Air Conditioner	A003_Samsung	23500	11	258500
15	Refrigerator	R003_Onida	23300	4	93200

```
>>> df1 = dfN.groupby(['Category_Name', 'Item_Num']).cumsum()
>>> df1.sum()          # Summing of all numeric columns or series
Unit_Price          476999
Sales_Quantity         106
Total_Amount       3078146
dtype: int64
# Summing a particular column or series
>>> dfN['Total_Amount'].groupby(dfN['Category_Name']).sum()
Category_Name
Air Conditioner       779200
Microwave             327550
Refrigerator          449200
Television            850200
Washing Machine       671996
Name: Total_Amount, dtype: int64
# Finding the mean of all series of a DataFrame
>>> print (dfN.groupby('Category_Name').mean())
                    Unit_Price  Sales_Quantity  Total_Amount
Category_Name
Air Conditioner   30233.333333        9.333333  259733.333333
Microwave         20366.666667        5.666667  109183.333333
Refrigerator      30133.333333        4.666667  149733.333333
Television        42075.000000        5.250000  212550.000000
Washing Machine   22166.333333        8.666667  223998.666667
```

Number of of unique column values per group

We can find the unique column values per group by using the **.nunique()** method of groupby() method. For example to find the number of unique column values per each **Category_Name** for **Sales_Quantity** column of DataFrame dfN:

```
>> dfN.groupby('Category_Name')["Sales_Quantity"].nunique()
Category_Name
Air Conditioner     2
Microwave           3
Refrigerator        2
Television          3
Washing Machine     3
Name: Sales_Quantity, dtype: int64
```

Using the .agg( ) Method with groupby

When we have a groupBy object, we may choose to apply one or more functions to one or more columns, even different functions to individual columns. We can aggregate by multiple functions using the **.agg()** method. Simple pass a list of the functions that you would like to apply to your dataset. The **.agg()** method allows us to easily and flexibly specify these details.

It takes as arguments the following:

- list of function names to be applied to all selected columns
- tuples of (colname, function) to be applied to all selected columns
- dict of (df.col, function) to be applied to each df.col
- Apply >1 functions to selected column(s) by passing names of functions to agg() as a list

```
# Summing a Series
>>> dfN['Total_Amount'].groupby(dfN['Category_Name']).agg('sum')
```

Category_Name

Air Conditioner	779200
Microwave	327550
Refrigerator	449200
Television	850200
Washing Machine	671996

Or

```
>>> dfN.groupby(dfN['Category_Name']).agg({'Total_Amount':['sum']}).reset_index()
```

	Category_Name	Total_Amount
		sum
0	Air Conditioner	779200
1	Microwave	327550
2	Refrigerator	449200
3	Television	850200
4	Washing Machine	671996

Example: Write the command to find the Category_Name wise aggregates applying multiple aggregation functions like count, min, mean and max for Total_Amount.

```
# Finding the maximum Unti_Price of each Category_Name
# Apply min, mean, max and max to Total_Amount grouped by Category_Name
>>> dfN['Total_Amount'].groupby(dfN['Category_Name']).agg(['min', 'mean', 'max'])
Or
>>> dfN.groupby('Category_Name').Total_Amount.agg(['count','min', 'mean', 'max'])
```

	count	min	mean	max
Category_Name				
Air Conditioner	3	258500	259733.333333	262200

Microwave	3	75000	109183.333333	143750
Refrigerator	3	93200	149733.333333	262800
Television	4	130000	212550.000000	329000
Washing Machine	3	38796	223998.666667	391200

Example **Write the command to create a hierarchical index to find minimum and maximum to all numeric columns for DataFrame dfN.**

```
# Apply min and max to all numeric columns of dfN grouped by Category_Name
# A Hierarchical index will be created
>>> dfN[['Unit_Price', 'Sales_Quantity', 'Total_Amount']].groupby(dfN['Category_Name']).agg(['min', 'max'])
```

	Unit_Price		Sales_Quantity		Total_Amount	
	min	max	min	max	min	max
Category_Name						
Air Conditioner	23500	43700	6	11	258500	262200
Microwave	13600	28750	4	8	75000	143750
Refrigerator	23300	43800	4	6	93200	262800
Television	27800	65800	4	8	130000	329000
Washing Machine	9699	32600	4	12	38796	391200

Example **Write the command to create a hierarchical index to find min and max to all numeric columns for DataFrame dfN by flipping the above layout of dfN, i.e., by moving whole levels of columns to rows.**

```
# We can call .stack() on the returned object!
>>> dfN[['Unit_Price', 'Sales_Quantity', 'Total_Amount']].groupby(dfN['Category_Name']).agg(['min', 'max']).stack()
```

		Unit_Price	Sales_Quantity	Total_Amount
Category_Name				
Air Conditioner	min	23500	6	258500
	max	43700	11	262200
Microwave	min	13600	4	75000
	max	28750	8	143750
Refrigerator	min	23300	4	93200
	max	43800	6	262800
Television	min	27800	4	130000
	max	65800	8	329000
Washing Machine	min	9699	4	38796
	max	32600	12	391200

Example **Create a histogram for the Total_Amount of dfN with bin size 5.**

```
# Creating histogram for Total_Amount
>>> import matplotlib.pyplot as plt
>>> dfN.hist(column="Total_Amount", bins=5)  # Plotting 5 bins
>>> plt.show()
```

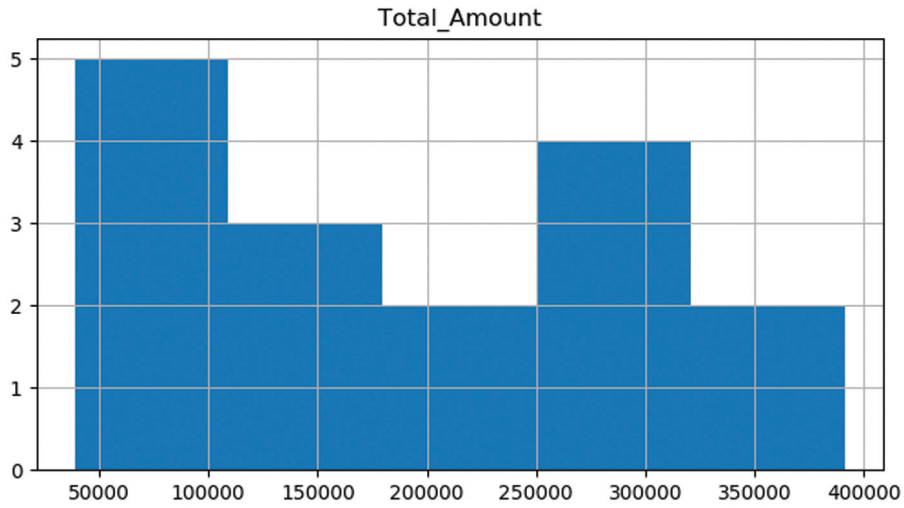


Figure 4.1 Histogram of a grouped data.

Grouping with Custom .apply() Functions

Just as you can apply custom functions to a column in your data frame, you can do the same with groups. As we know that **.apply()** method takes as argument the following:

- a general or user defined function
- any other parameters that the function would take

Let us apply the **.apply()** method with **groupby()** method to find the **Item_Num** wise total amount for each **Category Name** for the original DataFrame **df**. That is:

```
>>> df
```

	Category_Name	Item_Num	Unit_Price	Sales_Quantity
0	Television	T001_Panasonic	27800	8
1	Washing Machine	W003_Samsung	9699	4
2	Refrigerator	R001_LG	43800	6
3	Microwave	M001_LG	13600	8
4	Television	T002_Sony	42200	4
5	Air Conditioner	A001_LG	23500	11
6	Microwave	M002_Samsung	18750	4
7	Washing Machine	W001_IFB	32600	12
8	Television	T003_LG	32500	4

9	Refrigerator	R002_Samsung	23300	4
10	Air Conditioner	A002_Carrier	43700	6
11	Microwave	M003_LG	28750	5
12	Television	T004_Sony	65800	5
13	Washing Machine	W002_LG	24200	10
14	Air Conditioner	A003_Samsung	23500	11
15	Refrigerator	R003_Onida	23300	4

```
# User defined function to calculate the total amount
>>> def Calculate(ndf):
        return (ndf.Unit_Price * ndf.Sales_Quantity)
>>> df.groupby(['Category_Name', 'Item_Num']).apply(Calculate)
```

Category_Name	Item_Num		
Air Conditioner	A001_LG	5	258500
	A002_Carrier	10	262200
	A003_Samsung	14	258500
Microwave	M001_LG	3	108800
	M002_Samsung	6	75000
	M003_LG	11	143750
Refrigerator	R001_LG	2	262800
	R002_Samsung	9	93200
	R003_Onida	15	93200
Television	T001_Panasonic	0	222400
	T002_Sony	4	168800
	T003_LG	8	130000
	T004_Sony	12	329000
Washing Machine	W001_IFB	7	391200
	W002_LG	13	242000
	W003_Samsung	1	38796

```
dtype: int64
```

4.5 Data Transformation

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. Transformation on a group or a column returns an object that is indexed the same size of data is being grouped. And if you want to get a new value for each original row, use **transform()**. Thus, the transform should return a result that is the same size as that of a group chunk.

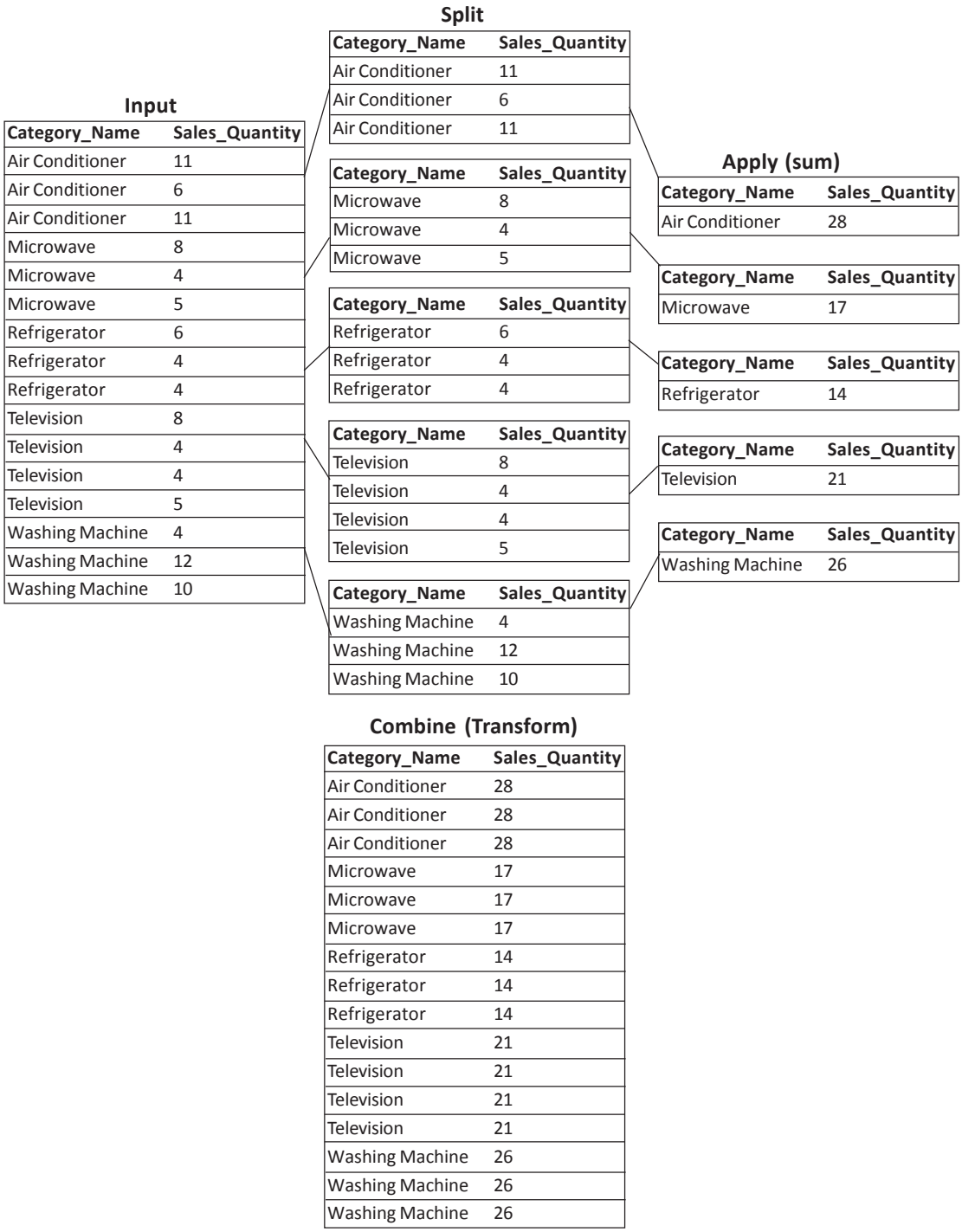
Before applying the **transform()** method, let us create a DataFrame **dfT** by sorting the previous DataFrame **dfN** with **Category_Name**.

```
>>> dfT = dfN.sort_values('Category_Name')     # a new dataframe dfT is created
>>> dfT
```

	Category_Name	Item_Num	Unit_Price	Sales_Quantity	Total_Amount
5	Air Conditioner	A001_LG	23500	11	258500
10	Air Conditioner	A002_Carrier	43700	6	262200
14	Air Conditioner	A003_Samsung	23500	11	258500
3	Microwave	M001_LG	13600	8	108800
6	Microwave	M002_Samsung	18750	4	75000
11	Microwave	M003_LG	28750	5	143750
2	Refrigerator	R001_LG	43800	6	262800
9	Refrigerator	R002_Samsung	23300	4	93200
15	Refrigerator	R003_Onida	23300	4	93200
0	Television	T001_Panasonic	27800	8	222400
4	Television	T002_Sony	42200	4	168800
8	Television	T003_LG	32500	4	130000
12	Television	T004_Sony	65800	5	329000
1	Washing Machine	W003_Samsung	9699	4	38796
7	Washing Machine	W001_IFB	32600	12	391200
13	Washing Machine	W002_LG	24200	10	242000

```
>>> dfT.groupby('Category_Name')["Sales_Quantity"].transform('sum')
5     28
10    28
14    28
3     17
6     17
11    17
2     14
9     14
15    14
0     21
4     21
8     21
12    21
1     26
7     26
13    26
Name: Sales_Quantity, dtype: int64
```

You will notice how this returns a different size data set from our normal **groupby()** functions. Instead of only showing the totals for 5 category names, we retain the same number of items as the original data set. That is the unique feature of using transform. Figure 4.2 shows the processes of **transform()** method.



Input

Category_Name	Sales_Quantity
Air Conditioner	11
Air Conditioner	6
Air Conditioner	11
Microwave	8
Microwave	4
Microwave	5
Refrigerator	6
Refrigerator	4
Refrigerator	4
Television	8
Television	4
Television	4
Television	5
Washing Machine	4
Washing Machine	12
Washing Machine	10

Split

Category_Name	Sales_Quantity
Air Conditioner	11
Air Conditioner	6
Air Conditioner	11

Category_Name	Sales_Quantity
Microwave	8
Microwave	4
Microwave	5

Category_Name	Sales_Quantity
Refrigerator	6
Refrigerator	4
Refrigerator	4

Category_Name	Sales_Quantity
Television	8
Television	4
Television	4
Television	5

Category_Name	Sales_Quantity
Washing Machine	4
Washing Machine	12
Washing Machine	10

Apply (sum)

Category_Name	Sales_Quantity
Air Conditioner	28

Category_Name	Sales_Quantity
Microwave	17

Category_Name	Sales_Quantity
Refrigerator	14

Category_Name	Sales_Quantity
Television	21

Category_Name	Sales_Quantity
Washing Machine	26

Combine (Transform)

Category_Name	Sales_Quantity
Air Conditioner	28
Air Conditioner	28
Air Conditioner	28
Microwave	17
Microwave	17
Microwave	17
Refrigerator	14
Refrigerator	14
Refrigerator	14
Television	21
Television	21
Television	21
Television	21
Washing Machine	26
Washing Machine	26
Washing Machine	26

Figure 4.2 Combined data using transform() method.

Example **Write the command to create new value mean of each row for columns "Unit_Price" and "Sales_Quantity".**

```
# Find the meanCreating histogram for Total_Amount
>>> dfT.groupby('Category_Name')["Unit_Price", "Sales_Quantity"].transform('mean')
```

	Unit_Price	Sales_Quantity
5	30233.333333	9.333333
10	30233.333333	9.333333
14	30233.333333	9.333333
3	20366.666667	5.666667
6	20366.666667	5.666667
11	20366.666667	5.666667
2	30133.333333	4.666667
9	30133.333333	4.666667
15	30133.333333	4.666667
0	42075.000000	5.250000
4	42075.000000	5.250000
8	42075.000000	5.250000
12	42075.000000	5.250000
1	22166.333333	8.666667
7	22166.333333	8.666667
13	22166.333333	8.666667

Example **A DataFrame df contains following data:**

```
>>> df
```

	Student_Name	Age	Gender	Test1	Test2	Test3
0	Aashna	16.0	F	7.6	8.5	7.6
1	NaN	NaN	NaN	NaN	NaN	NaN
2	Jack	16.0	M	8.6	NaN	NaN
3	Somya	17.0	F	6.5	7.9	8.8
4	Raghu	15.0	M	6.8	7.7	7.9
5	Mathew	16.0	M	9.2	9.0	NaN
6	Nancy	14.0	F	6.8	8.7	8.8

Write the command to find the mean of all numeric columns as per Gender.

```
# For DataFrame
>>> df = pd.read_csv('E:/IPSource_XII/IPXIIChap04/Student.csv')
```

```
# Create a function that
>>> def mean_age(ndf, col):    # here ndf is a dataframe and col is the column name
        # groups the data by a column (i.e., Gender) and returns the mean age per group
        return ndf.groupby(col).mean()
# Create a pipeline that applies the mean_age function to create a group according to column
>>> df.pipe(mean_age, col='Gender')
```

which will display the following output:

	Age	Test1	Test2	Test3
Gender				
F	15.666667	6.966667	8.366667	8.4
M	15.666667	8.200000	8.350000	7.9

4.6 .applymap() Function

The **.applymap()** function performs the specified operation for all the elements of a DataFrame. Remember that all columns (except index column) of the dataframe must be numeric type. The syntax is:

DataFrame.applymap(func)

Here,

- func: Python function, returns a single value from a single value.
- Returns: Transformed DataFrame.

For example, suppose we have a dataframe called tdf with following data:

```
>>> Test = {'A': [1, 2, 3, 4, 5, 6],
            'B': [6, 7, 8, 9, 10, 11]}
>>> tdf = pd.DataFrame(Test, columns=['A', 'B'])
>>> tdf
```

	A	B
0	1	6
1	2	7
2	3	8
3	4	9
4	5	10
5	6	11

Let us multiply by 2 to each elements of the dataframe **tdf**:

```
>>> func = lambda x: x*5
>>> tdf.applymap(func)
```

	A	B
0	5	30
1	10	35

```
2   15   40
3   20   45
4   25   50
5   30   55
```

Let us create another DataFrame **dfs** with following data and apply the **applymap()** function.

```
>>> import pandas as pd
>>> Data = {'Name': ['Aashna', 'Simran', 'Jack', 'Raghu', 'Somya', 'Ronald'],
            'English': [87, 64, 58, 74, 87, 78],
            'Accounts': [76, 76, 68, 72, 82, 68],
            'Economics': [82, 69, 78, 67, 78, 68],
            'Bst': [72, 56, 63, 64, 66, 71],
            'IP': [78, 75, 82, 86, 67, 71]}
>>> dfs = pd.DataFrame(Data, columns=['Name', 'English', 'Accounts', 'Economics', 'Bst', 'IP'])
>>> dfs
```

	Name	English	Accounts	Economics	Bst	IP
0	Aashna	87	76	82	72	78
1	Simran	64	76	69	56	75
2	Jack	58	68	78	63	82
3	Raghu	74	72	67	64	86
4	Somya	87	82	78	66	67
5	Ronald	78	68	68	71	71

```
>>> dfn = dfs.set_index(['Name'])          # Create a new DataFrame indexed permanently

>>> dfn
```

Name	English	Accounts	Economics	Bst	IP
Aashna	87	76	82	72	78
Simran	64	76	69	56	75
Jack	58	68	78	63	82
Raghu	74	72	67	64	86
Somya	87	82	78	66	67
Ronald	78	68	68	71	71

Now using the above DataFrame **dfn**, apply the function **.applymap()** to convert all numeric column cell value into float:

```
>>> dfn.applymap(float)
```

Name	English	Accounts	Economics	Bst	IP
Aashna	87.0	76.0	82.0	72.0	78.0
Simran	64.0	76.0	69.0	56.0	75.0
Jack	58.0	68.0	78.0	63.0	82.0
Raghu	74.0	72.0	67.0	64.0	86.0
Somya	87.0	82.0	78.0	66.0	67.0
Ronald	78.0	68.0	68.0	71.0	71.0

Example **Write the command to given an increment of 5% to all students to DataFrame df1 using applymap() function.**

```
# Create a function to increase 5% marks
>>> def increase5(x):
        return x + x*0.05
>>> dfn.applymap(increase5)          # Temporary inceases 5%
```

Name	English	Accounts	Economics	Bst	IP
Aashna	91.35	79.8	86.10	75.60	81.90
Simran	67.20	79.8	72.45	58.80	78.75
Jack	60.90	71.4	81.90	66.15	86.10
Raghu	77.70	75.6	70.35	67.20	90.30
Somya	91.35	86.1	81.90	69.30	70.35
Ronald	81.90	71.4	71.40	74.55	74.55

Or

```
# Use the lambda function
>>> dfn.applymap(lambda x:x + x*0.05)
```

4.7 Reindexing Pandas Dataframes

Reindexing in pandas is a process that changes the row labels and column labels of a DataFrame. This is core to the functionality of pandas as it enables label alignment across multiple objects, which may originally have different indexing schemes. This process of performing a reindex includes the following steps:

- Reordering existing data to match a new set of labels.
- Inserting NaN markers where no data exists for a label.
- Possibly, filling missing data for a label using some type of logic (defaulting to adding NaN values).

The syntax is:

DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=nan, limit=None, tolerance=None)

Here,

- labels : New labels/index to conform the axis specified by 'axis' to.
- index, columns : New labels / index to conform to. Preferably an Index object to avoid duplicating data.
- axis : Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).
- method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional.
- copy : Return a new object, even if the passed indexes are the same.
- level : Broadcast across a level, matching Index values on the passed MultiIndex level.
- fill_value : Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.
- limit : Maximum number of consecutive elements to forward or backward fill.
- tolerance : Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation abs(index[indexer] – target) <= tolerance.

Changing the order of the rows

To change the order (the index) of the rows of previous DataFrame **dfS**:

```
>>> dfs.reindex([5, 4, 3, 2, 1, 0])
```

So, the reindexed dataframe will be:

	Name	English	Accounts	Economics	Bst	IP
5	Ronald	78	68	68	71	71
4	Somya	87	82	78	66	67
3	Raghu	74	72	67	64	86
2	Jack	58	68	78	63	82
1	Simran	64	76	69	56	75
0	Aashna	87	76	82	72	78

```
>>> dfs.reindex([3, 4, 5, 2, 0, 1])
```

So, the reindexed dataframe will be:

	Name	English	Accounts	Economics	Bst	IP
3	Raghu	74	72	67	64	86
4	Somya	87	82	78	66	67
5	Ronald	78	68	68	71	71
2	Jack	58	68	78	63	82
0	Aashna	87	76	82	72	78
1	Simran	64	76	69	56	75

Change the order of the columns

To change the order (the index) of the columns of previous DataFrame **dfS**:

```
>>> ChangeColumns = ['Name', 'Accounts', 'English', 'Bst', 'Economics', 'IP']
>>> dfs.reindex(columns=ChangeColumns)
```

So, the reindexed dataframe will be:

	Name	Accounts	English	Bst	Economics	IP
0	Aashna	76	87	72	82	78
1	Simran	76	64	56	69	75
2	Jack	68	58	63	78	82
3	Raghu	72	74	64	67	86
4	Somya	82	87	66	78	67
5	Ronald	68	78	71	68	71

Reindexing with new index values

We can add new rows or columns by rendexing with new index values. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN. Let us create label index DataFrame using DataFrame dfS:

```
>>> ndf = dfs.set_index(['Name'])
>>> ndf
```

Name	English	Accounts	Economics	Bst	IP
Aashna	87	76	82	72	78
Simran	64	76	69	56	75
Jack	58	68	78	63	82
Raghu	74	72	67	64	86
Somya	87	82	78	66	67
Ronald	78	68	68	71	71

Now, create a new DataFrame called **df1** with a new row index '**Meghna**' using DataFrame **ndf**:

```
>>> df1 = ndf.reindex(['Aashna', 'Simran', 'Jack', 'Raghu', 'Meghna', 'Somya', 'Ronald'])
>>> df1
```

So, the reindexed dataframe will be:

Name	English	Accounts	Economics	Bst	IP
Aashna	87	76	82	72	78
Simran	64	76	69	56	75

Jack	58	68	78	63	82
Raghu	74	72	67	64	86
Meghna	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**
Somya	87	82	78	66	67
Ronald	78	68	68	71	71

Notice the above output where new indexes are populated with **NaN** values. Here, we can fill in the missing values using the parameter, **fill_value**:

```
>>> df1 = ndf.reindex(['Aashna', 'Simran', 'Jack', 'Raghu', 'Meghna', 'Somya', 'Ronald'], fill_value=73)
>>> df1
```

So, the reindexed dataframe will be:

	English	**Accounts**	**Economics**	**Bst**	**IP**
Name					
Aashna	87	76	82	72	78
Simran	64	76	69	56	75
Jack	58	68	78	63	82
Raghu	74	72	67	64	86
Meghna	**73**	**73**	**73**	**73**	**73**
Somya	87	82	78	66	67
Ronald	78	68	68	71	71

Simiarly, create a new column index '**TotalMarks**' using DataFrame **df1**:

```
>>> ChangeColumns = ['Name', 'Accounts', 'English', 'Bst', 'Economics', 'IP', 'TotalMarks']
>>> df1 = df1.reindex(columns=ChangeColumns)
>>> df1
```

So, the reindexed dataframe will be:

	English	**Accounts**	**Economics**	**Bst**	**IP**	**TotalMarks**
Name						
Aashna	87	76	82	72	78	**NaN**
Simran	64	76	69	56	75	**NaN**
Jack	58	68	78	63	82	**NaN**
Raghu	74	72	67	64	86	**NaN**
Meghna	73	73	73	73	73	**NaN**
Somya	87	82	78	66	67	**NaN**
Ronald	78	68	68	71	71	**NaN**

```
# To calculate the TotalMarks column values
>>> df1.TotalMarks = df1.Accounts + df1.English + df1.Bst + df1.Economics + df1.IP
```

```
>>> df1
```

Name	English	Accounts	Economics	Bst	IP	TotalMarks
Aashna	87	76	82	72	78	395
Simran	64	76	69	56	75	340
Jack	58	68	78	63	82	349
Raghu	74	72	67	64	86	363
Meghna	73	73	73	73	73	365
Somya	87	82	78	66	67	380
Ronald	78	68	68	71	71	356

4.8 Altering Labels or Chaning Column/Row Names

In pandas, there are two ways where one can change the column names of a pandas DataFrame. One way to rename columns in Pandas is to use **df.columns** from Pandas and assign new names directly. To demonstrate this, let us create a simple DataFrame with following:

```
>>> import pandas as pd
>>> Data1 = {'Customer_id': ['C_01', 'C-02' , 'C_03', 'C_04', 'C_05', 'C_06'],
'Product': ['Makeup', 'Mascara', 'Foundation', 'Lip Gloss', 'Eyeshadow', 'Eyeliner'],
'Charges': [650, 450, 550, 250, 150, 125]}
>>> dfm = pd.DataFrame(Data1, columns=['Customer_id', 'Product', 'Charges'])
>>> dfm
```

	Customer_id	Product	Charges
0	C_01	Makeup	650
1	C-02	Mascara	450
2	C_03	Foundation	550
3	C_04	Lip Gloss	250
4	C_05	Eyeshadow	150
5	C_06	Eyeliner	125

To change the columns of **dfm** DataFrame, we can assign the list of new column names to **dfm.columns** as:

```
>>> dfm.columns = ['CustomerID','Product_Choice','Fees']
>>> dfm
```

	CustomerID	Product_Choice	Fees
0	C_01	Makeup	650
1	C-02	Mascara	450
2	C_03	Foundation	550

3	C_04	Lip Gloss	250
4	C_05	Eyeshadow	150
5	C_06	Eyeliner	125

A problem with this approach to change column names is that one has to change names of all the columns in the DataFrame. This approach would not work, if we want to change just change the name of one column. Also, the above method is not applicable on index labels.

Another way to change column names in pandas is to use **.rename()** function. Using **.rename()** function, one can change names of specific column easily. And not all the column names need to be changed. One of the biggest advantages of using **.rename()** function is that we can use rename to change as many column names as we want.

The syntax is:

DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None)

Here,

- mapper, index and columns: Dictionary value, key refers to the old name and value refers to new name. Remember that only one of these parameters can be used at once.
- axis: int or string value, 0/'row' for Rows and 1/'columns' for Columns.
- copy: Copies underlying data if True.
- inplace: Makes changes in original Data Frame if True.
- level: Used to specify level in case data frame is having multiple level index.

Let us create a DataFrame as given below:

```
>>> import pandas as pd
>>> Teacher = {'TNO' : ['T01', 'T02', 'T03', 'T04', 'T05'],
'TNAME' : ['Rakesh Sharma', 'Jugal Mittal', 'Sharmila Kaur', 'Sandeep Kaushik', 'Sangeeta Vats'],
'TADDRESS' : ['245-Malviya Nagar', '34 Ramesh Nagar', 'D-31 Ashok Vihar', 'MG-32 Shalimar Bagh', 'G-35 Malviya Nagar'],
'SALARY' : [25600, 22000, 21000, 15000, 18900]}
>>> tdf = pd.DataFrame(Teacher, columns=['TNO', 'TNAME', 'TADDRESS', 'SALARY'])
>>> tdf
```

	TNO	TNAME	TADDRESS	SALARY
0	T01	Rakesh Sharma	245-Malviya Nagar	25600
1	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000
3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

Renaming a single column

For examle, to let us change a column name (**TNO** to **Teacher_No**) of above DataFrame tdf:

```
>>> tdf.rename(columns = {'TNO': 'Teacher_No'}, inplace=True)    # inplace=True to affect DataFrame
>>> tdf
```

	Teacher_No	TNAME	TADDRESS	SALARY
0	T01	Rakesh Sharma	245-Malviya Nagar	25600
1	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000
3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

From the above result, the first column is renamed as '**Teacher_No**'.

Renaing multiple columns

Let us change column names **TNAME** to **Teacher_Name**, **TADDRESS** to **Teacher_Address** and **SALARY** to **Income** of above DataFrame **tdf**:

```
>>> tdf.rename(columns = {'TNAME': 'Teacher_Name',
                          'TADDRESS' : 'Teacher Address',
                          'SALARY': 'Income'}, inplace=True)
```

From the above output:

- second column is renamed as '**Teacher_Name**'.
  third column is renamed as '**Teacher_Address**'.
- fourth column is renamed as '**Income**'.

So the resultant dataframe will be

```
>>> tdf
```

	Teacher_No	Teacher_Name	Teacher_Address	Income
0	T01	Rakesh Sharma	245-Malviya Nagar	25600
1	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000
3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

Renaming row index or row names

Another good thing about pandas rename function is that, we can also use it to change row indexes or row names. We just need to use index argument and specify, we want to change index not columns.

For example, to change row names 0 and 1 to 'zero' and 'one' in our **tdf** DataFrame, we will construct a dictionary with old row index names as keys and new row index as values.

```
>>> tdf.rename(index={0:'zero',1:'one'})
```

	Teacher_No	Teacher_Name	Teacher_Address	Income
zero	T01	Rakesh Sharma	245-Malviya Nagar	25600
one	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000

3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

Note that the above result does not change the DataFrame (**tdf**) row names permanenetly, because we omit the **inplace=True** option.

Renaming column name and row index simultaneously

With pandas' **rename()** function, one can also change both column names and row names simultaneously by using both column and index arguments to rename function with corresponding mapper dictionaries.

```
>>> tdf.rename(columns={'Teacher_No':'T_Number'},
                       index={0:'zero',1:'one'})
```

	T_Number	Teacher_Name	Teacher_Address	Income
zero	T01	Rakesh Sharma	245-Malviya Nagar	25600
one	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000
3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

Note that the above result does not change the DataFrame (**tdf**) row and column names permanently, because we omit the **inplace=True** option.

Using function input

Pandas **rename()** function can also take a function as input instead of a dictionary. For example, we can write a **lambda** function to take the current column names and consider only the first six characters for the new column names.

```
>>> tdf.rename(columns=lambda x: x[0:6])
```

	Teache	Teache	Teache	Income
0	T01	Rakesh Sharma	245-Malviya Nagar	25600
1	T02	Jugal Mittal	34 Ramesh Nagar	22000
2	T03	Sharmila Kaur	D-31 Ashok Vihar	21000
3	T04	Sandeep Kaushik	MG-32 Shalimar Bagh	15000
4	T05	Sangeeta Vats	G-35 Malviya Nagar	18900

Points to Remember

1. Pipe() function performs the custom operation for the entire dataframe.
2. The apply() method allows us to pass a function that will run on every value in a column.
3. The applymap() method applyes a function to each element of the DataFrame and returns a scalar to every element of a DataFrame.
4. The groupby() function splits the data into groups based on the levels of a categorical variable.
5. Reindexing in pandas is a process that changes the row labels and column labels of a DataFrame.

Solved Exercises

1. What is the use of describe() function?

Ans. The .describe() function is a useful summarisation tool that will quickly display statistics for any variable or group it is applied to.

2. Differentiate between .apply() and .applymap() functions.

Ans. The .apply() applies a function along any axis of the DataFrame whereas .applymap() apply a function to each element of DataFrame.

3. What is the default grouping is made using pandas groupby method?

Ans. The default the grouping is made via the index (rows) axis.

4. A dataframe dfW is given with following data:

	Age	Wage Rate
0	20	2.5
1	25	3.5
2	30	4.5
3	35	5.5
4	40	7.0
5	45	8.7
6	50	9.5
7	55	10.0
8	60	12.5

(*a*) Write a program using pipe() function to add 2 to each numeric column of dataframe dfW.
(*b*) Find the maximum value each column using apply() function.
(*c*) Find the row wise maximum value using apply() function.

Ans. (*a*)
```
# Wageplus.py
import pandas as pd
dfW = pd.DataFrame({'Age' : [20, 25, 30, 35, 40, 45, 50, 55, 60],
    'Wage Rate' : [2.5, 3.5, 4.5, 5.5, 7.0, 8.7, 9.5, 10.0, 12.5]},
    columns = ['Age', 'Wage Rate'])

def Add_Two(Data, aValue):
    return Data + aValue
print(dfW.pipe(Add_Two, 2))
```
(*b*)
```
dfW.loc[:, 'Age':'Wage Rate'].apply(max, axis=0)
```
(*c*)
```
dfW.loc[:, 'Age':'Wage Rate'].apply(max, axis=1)
```

5. A dataframe dfB is given with following data:

Itemno	ItemName	Color	Price
1	Ball Pen	Black	15.0
2	Pencil	Blue	5.5
3	Ball Pen	Green	10.5
4	Gel Pen	Green	11.0

5	Notebook	Red	15.5
6	Ball Pen	Green	11.5
7	Highlighter	Blue	8.5
8	Gel Pen	Red	12.5
9	P Marker	Blue	8.6
10	Pencil	Green	11.5
11	Ball Pen	Green	10.5

Answer the following questions using groupby function (assume that the dataframe name is dfB):

(*a*) Display Color wise item and price of each ItemName category.
(*b*) Find the maximum price of each ItemName.
(*c*) Find the minimum price of each ItemName.
(*c*) Count the number of items in each ItemName category.

Ans. (*a*) `dfX = dfB.groupby(['ItemName', 'Color'])`
`dfX.first()`
(*b*) `dfB.groupby('ItemName').Price.max()`
(*c*) `dfB.groupby('ItemName').Price.min()`
(d) `dfB.groupby('ItemName')['Color'].apply(lambda x: x.count())`

6. A dataframe contains following information:

	Customer	Region	Order_Date	Sales	Month	Year
0	K Books Distributers	East	2016-04-13	1256000	April	2016
1	GBC P House	South	2017-08-23	1359000	August	2017
2	S Books Store	North	2016-10-11	1670000	October	2016
3	TM Books	West	2019-08-25	1490000	August	2019
4	IND Books Distributors	North	2017-09-04	1560000	September	2017
5	Aniket Pustak	West	2018-05-17	1180000	May	2018
6	M Pustak Bhandar	South	2018-11-28	2100000	November	2018
7	BOOKWELL Distributors	North	2017-01-22	1630000	January	2017
8	Jatin Book Agency	West	2016-12-21	1380000	December	2016
9	New India Agency	South	2018-09-12	1730000	September	2018
10	Libra Books Distributors	East	2016-10-04	1210000	October	2016

Answer the following questions using groupby() function:

(*a*) Find the region wise average sales.
(*b*) Find the year wise average sales.
(*c*) Find the region wise aggregates of sales applying multiple aggregation functions like count, max, min and mean.
(*d*) What will be the outut of the following:
  (*i*) `dfN.groupby('Year').Sales.sum().round(decimals=2)`
  (*ii*) `dfN.groupby('Year').Sales.max().round(decimals=2)`
  (*iii*) `dfN.groupby('Region').Sales.min().round(decimals=2)`

Ans. For data:

```
dfN = pd.read_csv('E:/IPSource_XII/IPXIIChap02/Distributors.csv', index_col=0)
```

(*a*) `dfN.groupby('Region').Sales.mean().round(decimals=2)`

(*b*) `dfN.groupby('Year').Sales.mean().round(decimals=2)`

(*c*) `dfN.groupby('Region').Sales.agg(['count', 'max', 'min', 'mean']).round(decimals=2)`

(*d*) (*i*)
```
Year
2016    5516000
2017    4549000
2018    5010000
2019    1490000
Name: Sales, dtype: int64
```

(*ii*)
```
Year
2016    1670000
2017    1630000
2018    2100000
2019    1490000
Name: Sales, dtype: int64
```

(*iii*)
```
Region
East     1210000
North    1560000
South    1359000
West     1180000
Name: Sales, dtype: int64
```

Review Questions

1. What is the use pandas groupby() method?
2. Define apply() function with an example.
3. A dataframe called dfD is given with following data set:

	Quantity	Unit Price
0	12000	200
1	6500	180
2	500	250
3	500	350
4	13000	120
5	8800	130
6	2400	120
7	8000	170
8	8500	130
9	450	142

Write a command to find the Total Price (Quantity * Unit Price) using the lambda function.

4. A dataframe dfT is given with following data:

	Name	Qualification	Experience
0	Ms. Mittal	Masters	8
1	Minu Arora	Graduate	11
2	Sharmila Kaur	Post Graduate	7
3	Sangeeta Vats	Masters	9
4	Ramesh Kumar	Graduate	6
5	Jatin Ghosh	Post Graduate	8
6	Yash Sharma	Masters	10

Write the command for the following using pandas groupby() function :

(*a*) Find the average experience for each qualification.

(*b*) Find the total experience for each qualification.

(*c*) Find the average experience for each qualification and name.

5. A sample dataset is given with four quarter sales data for five employees:

Name of Employee	Sales	Quarter	State
R Sahay	125600	1	Delhi
George K	235600	1	Tamil Nadu
Jaya Priya	213400	1	Kerala
Manila Sahai	189000	1	Haryana
Ryma Sen	456000	1	West Bengal
Manila Sahai	172000	2	Haryana
Jaya Priya	201400	2	Kerala
George K	225400	2	Tamil Nadu
R Sahay	140600	2	Delhi
Ryma Sen	389000	2	West Bengal
Jaya Priya	242100	3	Kerala
George K	262000	3	Tamil Nadu
Ryma Sen	339000	3	West Bengal
Manila Sahai	228000	3	Haryana
R Sahay	193100	3	Delhi
George K	292000	4	Tamil Nadu
Manila Sahai	278000	4	Haryana
Jaya Priya	282100	4	Kerala
Ryma Sen	369000	4	West Bengal
R Sahay	233100	4	Delhi

Write the command for the following using pandas groupby() function (assume that the dataframe name is dfQ):

(*a*) Find the total sales of each employee.

(*b*) Find the state wise total sales.

(*c*) Find the total sales by both employees wise and state wise.

(*d*) Find the maximum individual sale by state.

(*e*) Find the employee name wise aggregates of sales applying multiple aggregation functions like count, max, min and mean.

(*f*) Find the output of the following:

(*i*) dfQ.groupby('Name of Employee').Sales.mean()

(*ii*) dfQ.groupby('State').Sales.mean()

6. A sample dataset is given with different columns as given below:

Item_ID	ItemName	Manufacturer	Price	CustomerName	City
PC01	Personal Computer	HCL India	42000	N Roy	Delhi
LC05	Laptop	HP USA	55000	H Singh	Mumbai
PC03	Personal Computer	Dell USA	32000	R Pandey	Delhi
PC06	Personal Computer	Zenith USA	37000	C Sharma	Chennai
LC03	Laptop	Dell USA	57000	K Agarwal	Bengalur
AL03	Monitor	HP USA	9800	S C Gandhi	Delhi
CS03	Hard Disk	Dell USA	5400	B S Arora	Mumbai
PR06	Motherboard	Intel USA	17500	A K Rawat	Delhi
BA03	UPS	Microtek India	4300	C K Naidu	Chennai
MC01	Monitor	HCL India	6800	P N Ghosh	Bengalur

Write the command for the following (assume that the dataframe name is dfA):

(*a*) Find city wise total price.

(*b*) Find manufacturer wise total price.